Parallel Image Renderer in C++ with OpenMP

Giovanni Colombo
Mat. 7092745
giovanni.colombo@edu.unifi.it

Abstract

This report describes the development of a simple Image Renderer in C++ and quantitatively compares its sequential and parallel implementations. The analysis demonstrates the significant potential of parallel computing and multithreading using the OpenMP API. Performance measurements including speedup and efficiency are conducted across various environmental configurations, hyperparameters, and different combinations of directives and clauses. Midterm Assignment for the Parallel Programming for Machine Learning course, taught by Professor Marco Bertini at the University of Florence, Italy.

1. Introduction

This report presents the development of a simple *Image Renderer* that generates and projects 3D circles onto a 2D rectangular surface, simulating a canvas. The renderer handles overlapping circles by mixing their colors properly, at different levels of depth. For simplicity, we constrain the 2D surface to be square, with dimensions *(canvas_size x canvas_size)*.

Each generated circle has six attributes:

- 2D coordinates (x, y): random values in range [0, canvas_size]
- Depth coordinate (z): random value in range [0, 1000]
- Radius: random value in range [10, 50]
- Color: *struct* defined by its (r,g,b) components, which are random values in range [0, 1]
- Transparency (alpha): random value in range [0.1, 0.5]

The z-coordinate, which indicates the depth in the 3D space, represents the distance of circles from the 2D projection surface. This allows defining an ordering of projection levels, which will be the basis for getting a coherent rendering. Determining the correct circle ordering is crucial for a successful rendering, as each circle is semitransparent. Therefore, a circle's color will not completely cover the color of an underlying circle (Figure 1). Instead, *color blending* occurs according to the *alpha blending* formula:

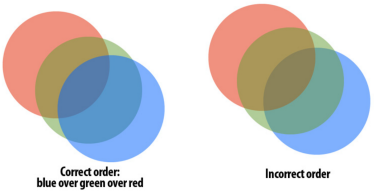

Figure 1: Visual representation of alpha blending.

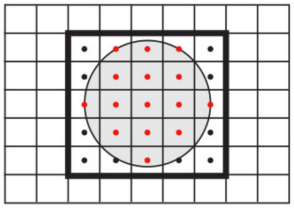

Figure 2: Visual representation of the pixel-circle intersection rule. Black dots represent pixels outside the circle; red dots represent pixels belonging to the circle.

$$Value_{color} = (1 - \alpha)Value^{A}_{color} + \alpha Value^{B}_{color}$$

where $color = \{r, g, b\}$, and A and B are two consecutive circles after ordering, with A being closer to the 2D surface than B.

So, after creating the circles, it is essential to perform two key operations **for each pixel** on the 2D surface: first, ordering the circles that would project onto that pixel according to their z-coordinate, and then calculating the pixel's final color following the alpha blending logic.

A circle with center coordinates (xc, yc) belongs to a pixel with coordinates (x, y) if the distance between these two points is less than the radius of the circle (Figure 2).

The final output of the *Image Renderer* is a *(canvas_size x canvas_size)* image showing semitransparent colored circles, which overlap each other (Figure 3).

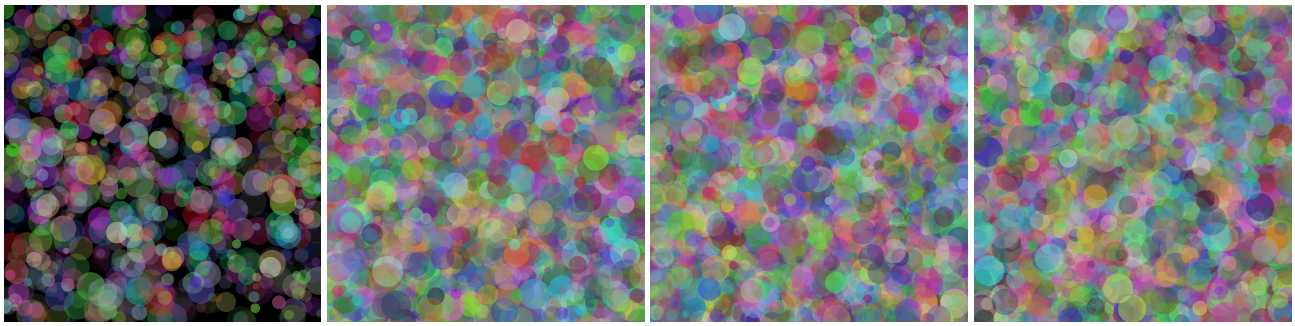

Figure 3: Rendering of 1000, 10000, 50000 and 100000 circles projected on a (1024 x 1024) canvas.

2. Method and Code

From an operational perspective, an *Image Renderer* as described in Section 1 essentially performs two main operations:

1. *Sorting* circles based on their distance from the 2D surface, indicated by their z-coordinate;
2. *Alpha blending* on individual pixels to determine the final color distribution.

Since circles are randomly positioned in space, different pixels on the 2D surface will belong to different numbers of circles with different colors, resulting in varying final pixel colors. Importantly, whether a circle belongs to a pixel (and vice versa) is a condition that depends **solely** on that pixel.

It is then crucial to note that both operations can be performed **independently** on each of the $canvas\_size^2$ pixels of the 2D surface, making this *Image Renderer* an *embarrassingly parallel* problem. This characteristic promises an effective code parallelization with OpenMP.

2.1. Hardware and Software setup

All tests were performed on an MSI Pulse GL76, with an Intel Core i7-12700H 14-core processor running at 2.7 GHz base frequency, and 16GB of RAM.

2.2. Code general structure

The project is built around a main **Renderer** class, which is the core of the implementation. This class manages all the operations for circle rendering. It defines the 2D surface (named "canvas") through its dimensions (for simplicity, `width = height = canvas_size`), a collection of circles to render stored in a vector, and the canvas itself implemented as a linear array of pixels, where each pixel is represented by a `Color` object, which defines its RGB values.

Operationally, the class implements three fundamental private methods: `isPixelInCircle`, which determines if a point belongs to a circle, `processPixel`, which calculates the final color of a pixel considering all circles containing it, and `alphaBlending`, which handles color composition considering circle transparency effects.

The class' public interface includes the two methods `renderSequential` and `renderParallel` that implement the sequential and parallel versions of the program respectively.

The outputs of these methods are two structs, `SequentialResult` and `ParallelResult`, containing essential data and measurements for the quantitative characterization of each run.

The `main.cpp` file allows defining several constants, to enable multiple combinations in the same run:

- `CANVAS_SIZES`: 2D surface dimensions {256, 512, 1024}
- `NUM_CIRCLES`: number of generated circles {1000, 5000, 10000, 20000, 50000, 100000}
- `NUM_THREADS`: number of threads created during fork {2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64}
- `BLOCK_SIZES`: number of blocks/chunks for scheduling {16, 24, 32}

2.3. Sequential Implementation

The sequential implementation lies in the `renderSequential` method.

```
sort(circles.begin(),
    ↪ circles.end(),[](const Circle& c1,
    ↪ const Circle& c2) { return c1.z >
    ↪ c2.z; });

for(int y = 0; y < height; y++) {
    for(int x = 0; x < width; x++) {
        for (const Circle& circle :
            ↪ circles) {
          if (isPixelInCircle(x, y,
              ↪ circle)) {
```

```
                Color finalColor;
                finalColor =
                    ↪ alphaBlending(circle.color,
                    ↪ finalColor,
                    ↪ circle.alpha);
            }
        }
        canvas[y * width + x] = finalColor;
    }
}
```

The process consists of two main steps:

1. **Global Circle Sorting**:
    - Sorts all circles by their z-coordinate in descending order
    - Uses the standard C++ `sort` algorithm
    - Complexity: $O(num\_circles * log(num\_circles))$
2. **Pixel Processing**:
    - Nested loops iterate over each pixel (x, y) in the canvas
    - For each pixel, checks if the pixel lies within the circle using Euclidean distance, and then applies alpha blending for overlapping circles
    - Complexity: $O(canvas\_size^2 * num\_circles)$

The total complexity is therefore: $O(num\_circles * log(num\_circles) + canvas\_size^2 * num\_circles)$

For realistic scenarios, the pixel processing phase dominates the execution time due to its higher complexity and the intensive floating-point operations required for color blending.

2.4. Parallel Implementation

The parallel implementation of the Image Renderer is implemented in the *renderParallel* method.

While the sorting algorithm's complexity remains the same as in the sequential implementation, as it is applied globally, the color blending operation represents the true bottleneck of the sequential implementation. Therefore, this operation presents the main opportunity to improve program performance.

This operation can be performed **independently** on each pixel of the 2D surface, making it possible to leverage OpenMP directives to execute these operations simultaneously on multiple pixels across `num_threads` threads, thus significantly reducing the total Execution Time.

By placing a simple `#pragma omp parallel` directive before the nested loops and `#pragma omp for` at the *outer* loop level, work can be easily distributed among a team of threads, in a number that is automatically decided. This initial implementation provides a baseline for parallel performance.

```
sort(circles.begin(),
    ↪ circles.end(),[](const Circle& c1,
    ↪ const Circle& c2) { return c1.z >
    ↪ c2.z; });

#pragma omp parallel
    #pragma omp for
    for(int y = 0; y < height; y++) { ... }
```

These two directives were then merged into the combined `#pragma omp parallel for` construct, which provides a more concise and potentially more efficient implementation. This refinement eliminates potential overhead from separate parallel region creation and work distribution directives. Then, to investigate the impact of the number of forked threads on the general performance, the `num_threads` clause was introduced. This addition allows control over the number of parallel threads.

```
#pragma omp parallel for
    ↪ num_threads(NUM_THREADS)
    for(int y = 0; y < height; y++) { ... }
```

Having two nested for loops at the pixel processing level, another interesting clause that could be evaluated was the `collapse(2)` clause, which merges the two nested loops into a single parallel region, potentially providing better load balancing and work distribution among threads.

```
#pragma omp parallel for collapse(2)
    ↪ num_threads(NUM_THREADS)
    ↪ schedule(static/dynamic, block_size)
    for(int y = 0; y < height; y++) { ... }
```

The final optimization phase explored different scheduling strategies and their impact on performance. OpenMP provides various scheduling options, including `static` and `dynamic` approaches, each with its own `block_size` parameter. This exploration aims to find the optimal balance between work distribution overhead and effective parallel execution.

```
#pragma omp parallel for
    ↪ num_threads(NUM_THREADS)
    ↪ schedule(static/dynamic, block_size)
    for(int y = 0; y < height; y++) { ... }
```

3. Performance Analysis and Results

Figure 4 clearly demonstrates how the OpenMP multithreading API can be efficient even in its simplest implementation.

The graph compares execution times between sequential and parallel implementations (just one line of code:

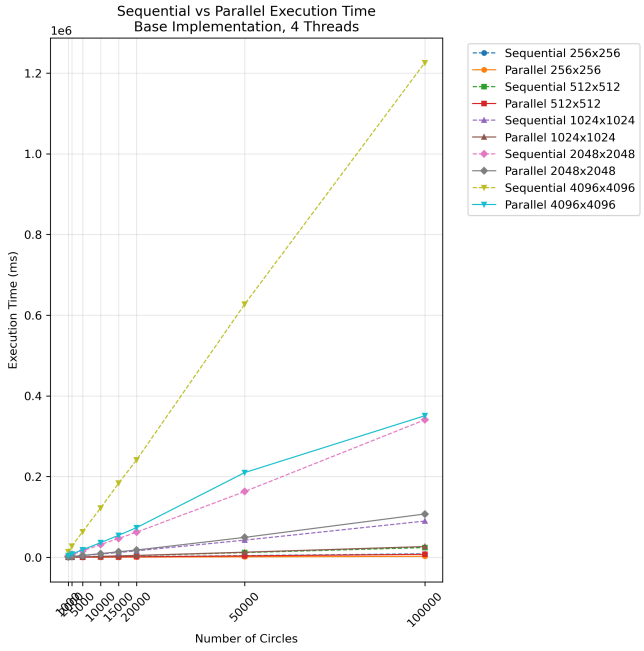

Figure 4: Execution time analysis of the parallel implementation for different canvas sizes and increasing number of circles.

`#pragma omp parallel for num_threads(4))` across different canvas sizes and number of circles. A nearly linear relationship between the number of circles and execution time is observed for each canvas size. This behavior reflects the computational complexity of the algorithm, which must process each pixel of the canvas for every single generated circle.

3.1. Thread Scaling Analysis

The parallel implementation of the Image Renderer shows interesting patterns, illustrated in Figures 5 and 6. Generally, different implementation strategies show similar performances, with important improvements over the sequential implementation: using from 12 to 24 threads achieves a speedup of approximately 6-7x while still maintaining a decent efficiency (50-40%). These values generally go down as the number of circles increases.

One of the most notable aspects is the characteristic behavior of speedup, which shows a clear tendency towards saturation as the number of threads increases. Initially, increasing the number of threads produces a *nearly linear* growth of speedup. However, beyond 12 threads, the curve begins to flatten, reaching a plateau around 6-7x for the best configurations. This behavior is correlated with a consistent decrease of efficiency from high (about 80-90% with few threads) to very low values (below 20%) with 32-64 threads. This phenomenon is a clear demonstration of *Amdahl's Law*: even in a theoretically highly parallelizable problem like this one, some parts of the program are inherently sequential (such as the initial sorting phase), and the parallelization overhead becomes increasingly significant as the number of threads increases. Consequently, there is a limit to the performance achievable through parallelization, and adding more threads beyond that limit may even lead to worse performances.

3.1.1 False Sharing Analysis

Interestingly, the implementation built to address false sharing through padding (at 32 or 64 bytes) shows significantly worse performance, even compared to the inner loop `for` directive implementation. This suggests that the overhead introduced to manage false sharing outweighs the benefits of reducing this phenomenon.

3.2. Scheduling Strategy Evaluation

As demonstrated in Figures 7 and 8, `dynamic` scheduling generally shows slightly better performance than `static` scheduling, with consistently lower execution times and higher speedups. The block size has a relatively minor impact on performance. This advantage of dynamic scheduling can be explained by analyzing the actual workload distribution in the renderer. In theory, each pixel requires the same computational work: checking intersections with every circle and calculating the final color. However, the workload varies significantly across different pixels, because some pixels fall into more circles than others, requiring more alpha blending operations than others. Static scheduling assigns fixed rows of the canvas to each thread, which can lead to unbalanced workload. Dynamic scheduling, on the other hand, ensures a more uniform work distribution during execution.

4. Conclusion

This report presented the development of an Image Renderer in its sequential and parallel implementations using OpenMP. The performance analysis demonstrated that parallelization significantly improved performances, achieving linear speedup using a limited number of threads, and reaching a peak of 7x with 16-24 threads while maintaining acceptable efficiency levels (around 40-50%). Performances obtained using configurations with too many threads demonstrated the existence and the effects of Amdahl's Law.

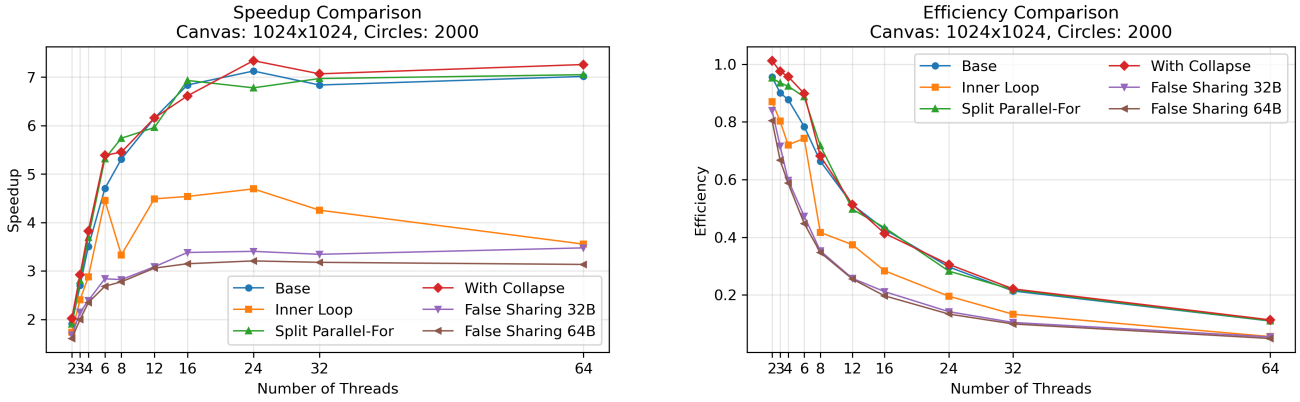

Figure 5: Performance analysis of different parallel implementations, with 2000 circles on a 1024x1024 canvas. Comparison of speedup and efficiency metrics.

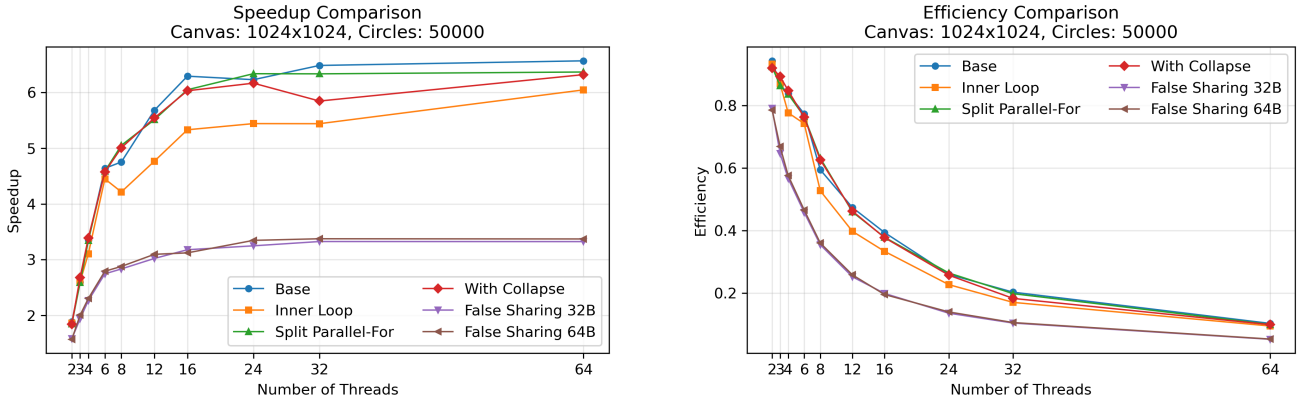

Figure 6: Performance analysis of different parallel implementations, with 50000 circles on a 1024x1024 canvas. Comparison of speedup and efficiency metrics.

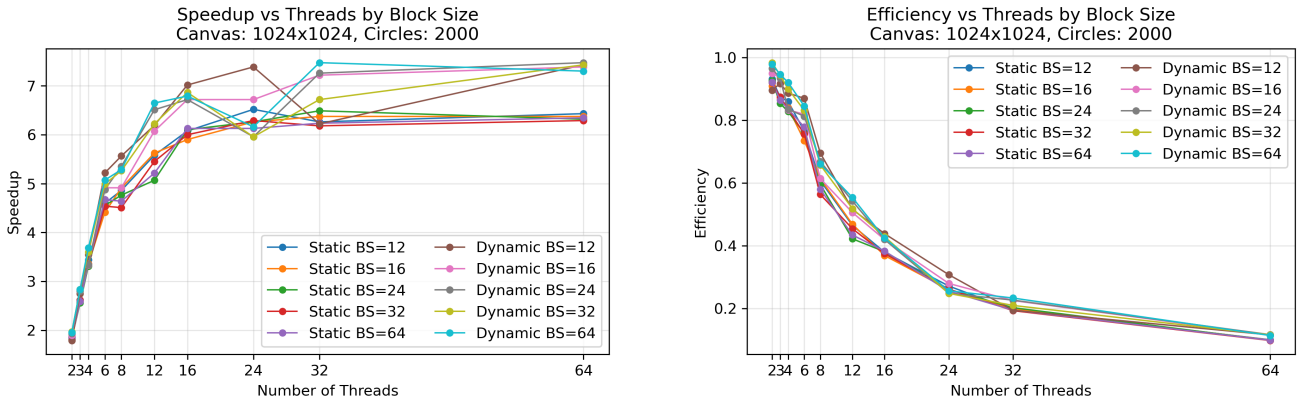

Figure 7: Performance analysis of different scheduling strategies, with 2000 circles on a 1024x1024 canvas. Comparison of speedup and efficiency metrics.

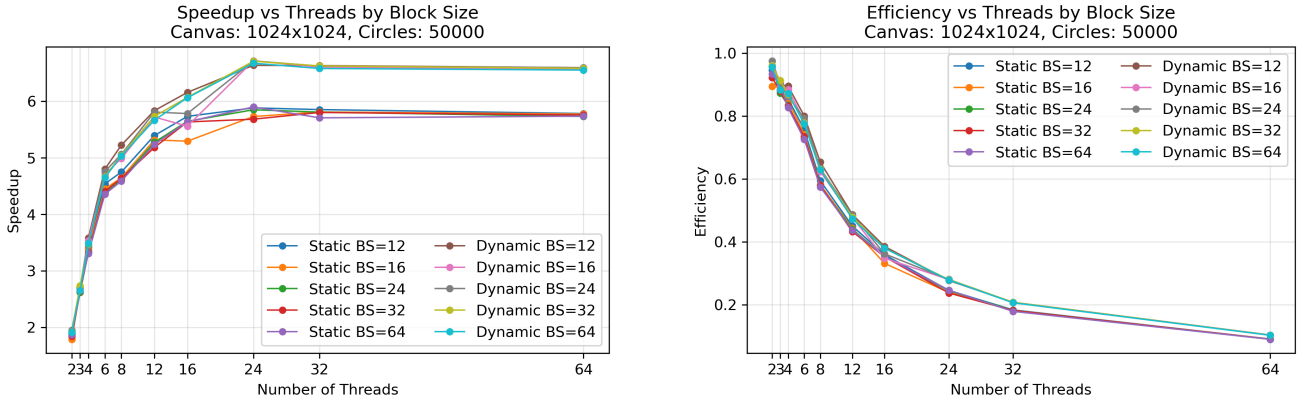

Figure 8: Performance analysis of different scheduling strategies, with 50000 circles on a 1024x1024 canvas. Comparison of speedup and efficiency metrics.